Principe et Algorithme Cryptographiques : DS n°2

première session

Université de Lille-1
FIL — PAC

20 mai 2016

Ce sujet est composé de 3 pages. Vos réponses doivent **absolument** être justifiées. Les différentes parties sont indépendantes et ne sont pas classées par ordre de difficulté. Toutes contiennent des questions de difficultés variés.

1 Faille d'OpenSSL dans Debian

Voici à peu près ce qui se passe quand on invoque la bibliothèque OpenSSL en réclamant la production d'une paire de clefs RSA :

1. Un générateur pseudo-aléatoire de qualité cryptographique est initialisé de manière imprévisible.
2. Deux nombres premiers aléatoires, p et q sont générés.
3. La clef publique (n, e) et la clef secrète d sont produites à partir de p et q.

À toute fin utile, on peut imaginer que la procédure de génération d'un nombre premier aléatoire de k bits fonctionne de la façon suivante :

— Générer un nombre (impair) de k bits uniformément au hasard
— Tant qu'il n'est pas premier, revenir à l'étape précédente.

▷ **Question 1 :** Quel est l'ordre de grandeur du temps nécessaire à la production d'un nombre premer aléatoire de 1000 bits (en 2016, sur une machine normale) ?

Environ 1 seconde.

▷ **Question 2 :** En gros, à combien d'itérations de la procédure peut-on s'attendre avant d'obtenir le résultat ?

Le théorème des nombres premiers dit que dans l'intervalle $[0; 2^{1000}]$, il y a à peu près $2^{1000}/\ln 2^{1000}$ nombres premiers. Ceci signifie que si on en prend un nombre au hasard, la probabilité qu'il soit premier est d'environ $1/(1000 \ln 2)$. En première approximation, on peut dire que ceci fait environ $1/1000$.

▷ **Question 3 :** Quel est l'ordre de grandeur du nombre de bits pseudo-aléatoires nécessaires pour produire une paire de clefs RSA de 2048 bits par cette procédure ?

Il faut environ 1000 essais pour produire un nombre premier, chaque essai « consomme »1000 bits pseudo-aléatoires, et il faut 2 nombres premiers. Total : 2 millions de bits.

En septembre 2006, un programmeur de la distribution Debian de Linux a modifié le code source d'OpenSSL pour éviter un avertissement lors de l'exécution d'un analyseur statique. Une conséquence inattendue de la modification, repérée en mai 2008, est que le générateur pseudo-aléatoire n'est initialisé qu'avec le `PID` du processus courant (sur Linux, c'est un nombre compris entre 0 et 2^{15}).

▷ **Question 4 :** J'ai généré une paire de clefs RSA avec la version 2006–2008 d'OpenSSL, et j'ai posté ma clef publique sur internet. Expliquer comment un adversaire pourrait tirer parti de la modification d'OpenSSL pour obtenir ma clef secrète.

En fait, le gros problème est que le PRNG ne peut plus produire que 2^{15} séquences de bits pseudo-aléatoires différentes. Donc, la génération de clef ne peut plus produire que 2^{15} paires de clefs différentes ! Ma clef publique est donc l'une des 32768 clefs publiques possibles. Un attaquant peut donc :

1. $PID \leftarrow 0$
2. Initialiser le PRNG avec le PID
3. Générer une paire de clefs avec la séquence pseudo-aléatoire produite
4. S'il trouve ma clef publique, il trouve du même coup ma clef privée, et c'est fini.
5. $PID \leftarrow PID + 1$, puis retourner à l'étape 2.

Ceci trouve ma clef secrète en moins d'une demi-journée sur un ordinateur portable.

2 Faiblesse de la fonction `rand()` en C

Cet exercice est une des missions « secrètes » du TP, que personne n'a fait. Considérons le programme suivant, en langage C.

```c
#include <stdint.h>
#include <stdio.h>

void encrypt(char *message, int message_size) {
  uint32_t K[4], IV[2];
  FILE *output;

  srand48( time(NULL) );

  K[0] = mrand48();
  K[1] = mrand48();
  K[2] = mrand48();
  K[3] = mrand48();

  IV[0] = mrand48();
  IV[1] = mrand48();
  IV[2] = mrand48();
  IV[3] = mrand48();

  printf("Randomly generated key : %x-%x-%x-%x\n', K[0], K[1], K[2], K[0])

  ciphertext = aes_128_CBC(plaintext, message_size, K, IV);

  output = fopen("ciphertext.bin", "w");
  fwrite(IV, 16, 1, output);
  fwrite(ciphertext, message_size, 1, output);
  fclose(output);
}
```

En gros, c'est une fonction qui génère une clef symétrique aléatoire, ainsi qu'un IV aléatoire. La clef est affichée (nous supposons qu'elle est écrite sur un bout de papier par l'utilisateur). Le message est chiffré puis écrit dans un fichier, précédé par l'IV.

Le générateur pseudo-aléatoire utilisé, la fonction `mrand48()`, fait partie de la norme POSIX. Elle doit être implémentée de la façon suivante :

```c
uint64_t rand48_state;

void srand48(uint32_t seed) {
  rand48_state = 0x330e + (seed << 16);
}

uint32_t mrand48() {
  rand48_state = (0x00000005deece66d * rand48_state + 11) & 0x0000ffffffffffff;
  return (rand48_state >> 16);
}
```

L'appel à `srand48()` permet d'entrer une « graine » dans le générateur pseudo-aléatoire. Chaque appel à `mrand48()` renvoie 32 bits pseudo-aléatoires, et met à jour l'état interne de la fonction. Le générateur tire son nom du fait que son état interne occupe 48 bits (en effet, les 16 bits de poids fort de la variable `rand48_state` sont toujours nuls).

▷ **Question 5 :** Cette manière d'initialiser le générateur pseudo-aléatoire vous semble-t-elle raisonnable ?

Non, ce n'est pas terrible. Le problème, c'est que l'heure qu'il est actuellement (le nombre de secondes écoulées depuis le 1er janvier 1970) est quelque chose de prévisible. Imaginons une interaction entre un client et un serveur. Si le serveur est à l'heure, alors le client (s'il est à l'heure lui aussi) pourrait connaître les nombres aléatoires générés par le serveur, et qui sont parfois censés rester privés.

▷ **Question 6 :** Décrivez une attaque très simple qui permet de déchiffrer le fichier `ciphertext.bin` en 2^{32} essais (sans connaître la clef, bien sûr).

C'est le même argument que dans la dernière question de la 1ère partie. Le générateur, lorsqu'il est initialisé, ne peut se trouver que dans 2^{32} états différents (toutes les valeurs possibles de la variable `seed` ligne 3). Une fois que son état est fixé, la séquence produite est toujours la même.

1. $seed \leftarrow 0$
2. `srand48(seed)`.
3. Invoquer `mrand48()` 4 fois pour générer la clef K.
4. Invoquer `mrand48()` 4 fois pour générer l'IV.
5. Si l'IV est celui qui est au début du fichier, alors utiliser la clef pour déchiffrer le fichier. C'est fini.
6. $seed \leftarrow seed + 1$, puis retourner à l'étape 2.

Ceci identifie la clef secrète à coup sûr en 2^{32} itérations au maximum.

▷ **Question 7 :** En terme d'arithmétique modulaire, quel est l'effet de l'opération AND de la ligne 8 ?

Calculer $y \leftarrow x$ & `0x0000ffffffffffff` revient à calculer $y \leftarrow x \bmod 2^{48}$. C'est une ruse d'implémentation couramment utilisée, car le AND est légèrement plus rapide que la division sur les CPU modernes.

▷ **Question 8 :** Supposons qu'on connaisse la valeur de la variable `rand48_state`, juste après l'exécution de la ligne 8 dans la fonction `mrand48()`. Expliquez comment on peut faire pour « remonter » efficacement à la valeur qu'avait cette variable *avant* l'exécution de la ligne 8.

Si on regarde bien, ce qui se passe c'est :

$$x' \leftarrow a \times x + b \bmod 2^{48}.$$

Du coup, pour retrouver x à partir de x', il faut résoudre cette équation linéaire modulo 2^{48}. On trouve :

$$x \leftarrow a^{-1} \times (x' - b) \bmod 2^{48},$$

où a^{-1} désigne l'inverse modulaire de a modulo 2^{48}.

▷ **Question 9 :** Lorsqu'on connaît le premier mot de 32 bits de l'IV, connaît-on entièrement l'état interne du générateur pseudo-aléatoire ? Sinon, que manque-t-il ?

La sortie de `mrand48()` révèle les 32 bits de poids fort de l'état interne. Les 16 bits de poids faible restent inconnus.

▷ **Question 10 :** Déduisez-en une attaque qui permet de déchiffrer le fichier `ciphertext.bin` en 2^{16} essais. Ceci est-il réalisable en pratique, sans moyens spéciaux ?

L'idée de l'attaque est la suivante :

1. Lire l'IV dans le fichier.
2. Initialiser $lsb \leftarrow 0$
3. Calculer $state \leftarrow (IV[0] \ll 16) + lsb$, et définir ça comme état interne de `mrand48()`.
4. Invoquer `mrand48()` 3 fois. Si ça ne produit le reste de l'IV, alors passer à l'étape 6.
5. Incrémenter lsb et retourner à l'étape 3.
6. Résoudre 8 fois successivement l'équation, pour « rembobiner »8 fois l'invocation de `mrand48()`.
7. Invoquer `mrand48()` 4 fois pour produire la clef K. Déchiffrer le fichier avec.

Il est clair que la boucle des étapes 3–5 ne va faire que 2^{16} tours aux maximum (car il « manque »les 16 bits de poids faible de l'état interne).
Une fois qu'on a atteint l'étape 6, on est sûr d'avoir le même état interne que lors de la génération de la clef. Il suffit donc de « rembobiner »la fonction pour obtenir l'état interne initial, et pouvoir générer la clef de la même façon que l'utilisateur légitime.

3 Protocole « Secure Remote Password »

Le protocole « Secure Remote Password »(SRP) est un protocole d'échange de clef authentifié par mot de passe conçu dans les années 2000. Il s'agit d'une variante de l'échange de clef Diffie-Hellman. Par conséquent, on note p un grand nombre premier utilisé dans le protocole, et g un générateur raisonnable modulo p.

▷ **Question 11 :** Rappelez ce que sont les « PAKE »(Password Authenticated Key Exchange). Quelle type d'attaque visent-ils en particulier à éviter ?

Un PAKE permet à deux participants possédant en commun un secret cryptographique faible (le mot de passe) d'établit un secret partagé fort (une clef de l'AES aléatoire). Bien sûr, il faut que ça ne marche que si les deux participants ont vraiment le mot de passe en commun !
L'attaque principale à éviter est l'attaque par dictionnaire *offline* sur le mot de passe. Comme un mot de passe est court, il est envisageable de pouvoir énumérer un ensemble de chaine de bits (un dictionnaire) qui contienne la plupart des mots de passe. Il ne faut donc pas qu'un adversaire, passif ou actif, puisse lancer cette opération sans devoir interagir avec un des participants.

La caractéristique la plus frappante du protocole SRP, c'est que le serveur ne connaît pas le mot passe — et pourtant il peut vérifier que le client le connaît.

▷ **Question 12 :** Quel avantage est-ce que ceci procure par rapport à un protocole tel que SPEKE où le serveur possède le mot de passe ?

Si jamais le serveur était compromis (hacké, saisi par les autorités, admin fou, ...), alors le mot de passe n'est pas directement révélé.

Le protocole est composé de deux procédures : l'*enregistrement* (où un utilisateur s'inscrit sur le serveur) et la *connection* (où un utilisateur préalablement inscrit se connecte au serveur).

Enregistrement Pour s'enregistrer auprès du serveur, le client, dont l'identité est une chaine de bits I, et dont le mot de passe est une (courte) chaine de bits pwd, doit :

1. Calculer $x \leftarrow H(pwd)$
2. Calculer $v \leftarrow g^x \bmod p$ (c'est le « vérifieur »).
3. Envoyer (I, v) au serveur, qui les stocke.

▷ **Question 13 :** Le serveur apprend-il le mot de passe au terme de la procédure d'enregistrement ?

Non. Il apprend $g^{H(pwd)} \bmod p$. Pour obtenir pwd, il faudrait qu'il a) résolve un logarithme discret et b) inverse la fonction de hachage. Rien ne lui interdit, par contre, de tenter une recherche par dictionnaire sur le mot de passe...

▷ **Question 14 :** Supposons qu'un attaquant parvienne à intercepter la transmission pendant la phase d'enregistrement. Expliquez comment il pourrait s'y prendre pour tenter de découvrir le mot de passe.

La connaissance de v permet de tenter une attaque ar dictionnaire :

1. $pwd \leftarrow$ premier mot du dictionnaire.
2. Si $v = g^{H(pwd)} \bmod p$, alors renvoyer pwd.
3. $pwd \leftarrow$ mot suivant du dictionnaire. Retourner à l'étape 2.

Dans toute la suite, on suppose que la procédure d'enregistrement des utilisateurs est effectuée sur un canal sécurisé (par exemple, en accédant physiquement au serveur).

Connection La phase de connexion fonctionne de la façon suivante.

1. Le client génère un nombre aléatoire a, calcule $A \leftarrow g^a \bmod p$, puis envoie (I, A) au serveur (celui-ci interrompt le protocole si $A \equiv 0 \bmod p$).
2. Le serveur récupère le vérifieur v associé à l'identité I dans sa base de donnée. Il choisir un nombre aléatoire b, calcule $B \leftarrow v + g^b \bmod p$. Enfin, il envoie B au client (celui-ci interrompt le protocole si $B \equiv 0 \bmod p$).
3. Le client et le serveur calculent $u \leftarrow H(A \parallel B)$.
4. Le client calcule $S \leftarrow (B - g^x)^{a+ux} \bmod p$, puis $K \leftarrow H(S)$.
5. Le serveur calcule $S' \leftarrow (A \cdot v^u)^b \bmod p$, puis $K' \leftarrow H(S)$.
6. Le client calcule $M_1 \leftarrow H(I \| A \| B \| K)$, puis l'envoie au serveur (celui-ci interrompt le protocole si la valeur fournie est incorrecte).
7. Le serveur calcule $M_2 \leftarrow H(A \| M_1 \| K')$, puis l'envoie au client (celui-ci interrompt le protocole si la valeur fournie est incorrecte).

▷ **Question 15 :** Montrer que si les deux participants sont honnêtes, alors $S = S' = \left(g^b\right)^{a+ux}$.

Côté client :

$$\begin{aligned}(B - g^x)^{a+ux} &\equiv \left(v + g^b - g^x\right)^{a+ux} \bmod p \\ &\equiv \left(g^x + g^b - g^x\right)^{a+ux} \bmod p \\ &\equiv \left(g^b\right)^{a+ux} \bmod p.\end{aligned}$$

Côté serveur :

$$\begin{aligned}(a \cdot v^u)^b &\equiv (A \cdot v^u)^b \bmod p \\ &\equiv \left(g^a \cdot (g^x)^u\right)^b \bmod p \\ &\equiv \left(g^{a+ux}\right)^b \bmod p \\ &\equiv g^{b(a+ux)} \bmod p \\ &\equiv \left(g^b\right)^{(a+ux)} \bmod p.\end{aligned}$$

▷ **Question 16 :** Quel est l'intérêt des deux dernières étapes ?

Ces deux étapes servent à confirmer que les deux participants ont bien établi une clef de session K commune — donc qu'ils avaient bien le même mot de passe au début.
Du coup, si le client ne connait pas le mot de passe, il se fait bloquer à la fin de l'étape 6.

▷ **Question 17 :** La valeur de u peut-elle être apprise par un adversaire passif ?

Oui. Il suffit de connaître A et B pour pouvoir calculer u, or A et B sont transmis en clair.

▷ **Question 18 :** Expliquez pourquoi un attaquant qui espionnerait une exécution du protocole ne pourrait pas monter une attaque contre le mot de passe.

Le seul moment où le mot de passe intervient dans les transmissions du protocole, c'est lorsque $B \equiv v + g^b \bmod p$ est envoyé par le serveur au client. On a vu que connaître v permet une attaque par dictionnaire sur le mot de passe.
Mais ici, un espion n'apprend pas v, car g^b joue le rôle d'un masque jetable. En effet, b a été choisi aléatoirement, en secret, par le serveur. Du coup, si l'ordre de g est suffisamment grand, g^b peut prendre presque n'importe quelle valeur possible, et donc « masquer »la valeur de v.

▷ **Question 19 :** Alice exécute le protocole avec un serveur. Charlie connaît le mot de passe d'Alice (ils sont intimes), et en plus il espionne l'exécution du protocole. Peut-il obtenir la clef de session K ?

Non. Même en connaissant le mot de passe, Charlie ne peut pas obtenir ni S ni K.
En gros, la valeur de S commune à la fin du protocole est :

$$S = g^{ab} \times (g^b)^{ux}$$

S'il connaît le mot de passe, Charlie peut calculer x et v. Ceci lui permet de connaître g^b à partir de B. On a vu qu'il peut connaître u. Il peut donc calculer $(g^b)^{ux}$.
Mais il ne connaît ni a ni b, qui ont été choisi en secret par Alice et le serveur. Ceci l'empêche de calculer S ou S' comme le feraient Alice ou le serveur.
D'autre part, pour obtenir S, il lui manque g^{ab}. Il connaît g^a et g^b, mais calculer g^{ab} à partir ce ces deux valeurs, c'est précisément casser l'échange de clef Diffie-Hellman classique, qu'on suppose sûr.

▷ **Question 20 :** Un adversaire passif qui ne connait pas le mot de passe peut-il apprendre la clef de session K ?

Il semble que cet adversaire soit dans une situation plus difficile que celui qui connait le mot de passe. Donc on peut conclure assez facilement qu'il ne parviendra pas à obtenir K.